# Relazione Progetto APSD

Algoritmi di ordinamento paralleli

## 1. Introduzione

Il progetto consiste nell'implementazione di tre diversi tipi di algoritmi di ordinamento in versione parallela, ovvero:

- Parallel Bitonic Sort
- Parallel Odd Even Sort
- Parallel Quick Sort

Verranno spiegate le scelte implementative, come ad esempio:

- Suddivisione del lavoro tra i processi
- Tipologia di messaggistica utilizzata
- Dipenda tra i processi (Task Dependency Graph)

Inoltre, le versioni parallele implementate vengono messe a confronto con le rispettive controparti seriali per avere un'idea sulle ***performance*** delle versioni parallele, in particolare per il calcolo di:

- Speed-Up
- Total Parallel Overhead
- Efficienza
- Scalabilità

# 2. Parallel Odd Even Sort

L'Odd Even Sort è una variante del Bubble Sort. Come nel Bubble Sort, gli elementi di un array vengono confrontati a coppie e scambiati quando necessario. Tuttavia, questi confronti-scambi vengono eseguiti in due fasi: fase pari e fase dispari. Nella fase dispari, eseguiamo un Bubble Sort sugli elementi con indice dispari e nella fase pari eseguiamo un Bubble Sort sugli elementi con indice pari.

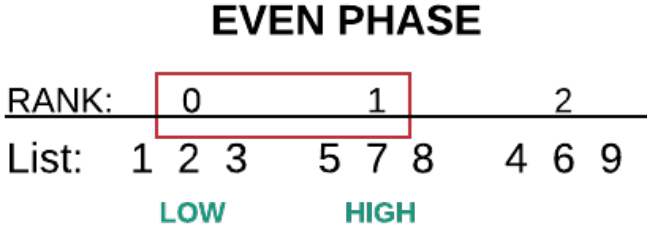
## 2.1 Parallelizzazione e suddivisione del lavoro tra i processi

Inanzitutto bisogna decidere come suddividere il lavoro tra i processi. Se abbiamo n elementi nella lista e p processi, allora naturalmente ogni processo riceve n/p elementi, che verranno ordinati ***indipendentemente*** da ogni processo. Per ordinare gli elementi locali in ogni processo, possiamo usare un algoritmo di ordinamento seriale veloce, come il Quick Sort.
A questo punto abbiamo processi indipendenti che contengono ciascuno una porzione locale di elementi ordinati: può iniziare il vero è proprio Odd Even Sort.

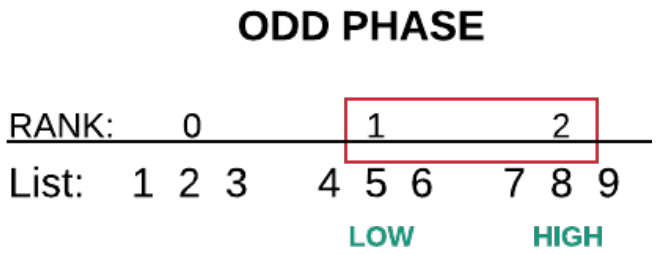
Ogni processo avrà un ***partner*** con cui **scambiare i dati locali ed effettuare un merge dei risultati**. In particolare se un processo ha rango *my_rank* dispari, il suo partner nella fase pari avrà rango *my_rank - 1* mentre il suo partner nella fase dispari avrà un rango *my_rank + 1*. Allo stesso modo, un processo il cui rango è pari avrà un partner con rango *my_rank + 1* nella fase pari e un partner con rango *my_rank + 1* in quella dispari.

Il merging funziona così:

- **Fase pari:** se abbiamo un processo di rango dispari, vogliamo che l'elenco locale di questo processo si unisca al suo partner nella fase pari in modo che il processo con rango dispari contenga gli elementi più grandi tra i due. Altrimenti, se il processo ha rango pari vogliamo che l'elenco locale di questo processo si unisca al suo partner nella fase pari in modo che il processo con rango pari contenga gli elementi più piccoli tra i due.



- **Fase dispari:** se abbiamo un processo di rango dispari, vogliamo che l'elenco locale di questo processo si unisca al suo partner nella fase dispari in modo che il processo con rango dispari contenga gli elementi più piccoli tra i due. Altrimenti, se il processo ha rango pari vogliamo che l'elenco locale di questo processo si unisca al suo partner nella fase dispari in modo che il processo con rango pari contenga gli elementi più grandi tra i due.




Il numero di fasi da eseguire per garantire un elenco ordinato dipende dal numero di processi utilizzati: se si utilizzano *p* processi, dopo *p* fasi l'elenco verrà ordinato. Effettuando il merging per tutte le *p* fasi, alla fine gli elementi risulteranno ordinati in modo crescente in processi di rango crescente: un processo con rango *i* avrà elementi più piccoli di un processo con rango *j*, per $i < j$.

## 2.2 Dipenda tra i processi

- **Fase pari:** nella fase pari ogni processo deve attendere il suo partener nella fase pari. In particolare un processo con rango pari deve attendere il processo con rango *my_rank + 1* e un processo con rango dispari deve attendere un processo con rango *my_rank – 1*.


- **Fase dispari:** nella fase dispari ogni processo deve attendere il suo partener nella fase dispari. In particolare un processo con rango pari deve attendere il processo con rango *my_rank – 1*e un processo con rango dispari deve attendere un processo con rango *my_rank + 1*.

## 2.3 Tipologia di messaggistica utilizzata

La tipologia di messaggistica utilizzata e la **blocking**, in quanto ogni processo prima di eseguire l'opportuna operazione di merging deve aver ricevuto preventivamente gli elementi del suo partner. A tale scopo è stata utilizza la funzione ***MPI_Sendrecv.***

Non avrebbe avuto senso quindi effettuare un invio/ricezione non bloccante, in quanto nel frattempo il processo non avrebbe comunque potuto svolgere nessuna operazione utile.

## 2.4 Performance

Tempi di esecuzione:

- Seriale

```
c:\Users\Utente\Documents\Università\Anno
ti\Progetto\Serial>.\serialOddEvenSort
TIME: 13.027 secondi
```

- Parallelo:

```
c:\Users\Utente\Documents\Università\Anno Accademico 2021
ti\Progetto\Parallel>mpiexec -n 4 parallelOddEvenSort.exe
TIME: 5.33144 secondi
```

Lo **Speed-Up** è pari a: $Ts/Tp$ = 13,027/5,33 = **2,45**. Lo Speed-Up è quasi pari a 3: essendo il numero di processi utilizzati pari a 4 si tratta di un buon risultato.

L'**efficienza** è pari a: $S/p$ = 2,45/4 = **0,61**. Si tratta di un buon valore in quanto la velocità è raddoppiata rispetto alla versione seriale (anche se sono stati utilizzati 4 processi).

L'**Overhead** è pari a: $To = p*Tp - Ts$ = 4*5,33 – 13,027 = **8,29**. Abbiamo quindi un overhead di 8 secondi. Il valore è abbastanza alto in quanto ogni processo deve comunicare, per ogni fase, il proprio chunck di elementi al proprio partener. Avendo eseguito i test su un numero totale di elementi elevato, tale comunicazione richiede del tempo, in quanto il chunck è di grandi dimensioni.

# 3. Parallel Quick Sort

L'algoritmo del Quick Sort è prettamente ricorsivo e segue i seguenti passaggi:

1. Scegliere un elemento pivot, di solito l'ultimo elemento della lista.
2. Posizionare, tramite un'operazione di comparazione e scambio, gli elementi più piccoli del pivot alla sua sinistra e i più grandi alla sua destra. Dopo questo passaggio il pivot si troverà nella giusta posizione finale e avremo due sotto-liste: quella alla sinistra del pivot e quella a destra.
3. Ritornare al passo 1, scegliendo come lista la sotto-lista sinistra e destra.

## 3.1 Parallelizzazione e suddivisione del lavoro tra i processi

Una volta eseguita la partizione, è possibile ordinare in parallelo diverse sezioni della lista (array) originario. Se abbiamo p processi, possiamo dividere una lista di n elementi in p sotto-liste: ad ogni processo viene assegnata una sezione della lista da ordinare, che il processo ordina ***indipendentemente*** utilizzando l'algoritmo del Quick Sort. Alla fine invia la sua sotto-sequenza ordinata per il merge.

Ad ogni passaggio (**step**) ogni processo invia la sua sotto-sequenza ordinata al suo ***vicino*** e viene eseguita un'operazione di merge. Per effettuare il merge di tutte le sotto-sequenze bisogna eseguire *log(n)* step, con n pari al numero di processi utilizzati, ovvero il numero di sotto-sequenze da unire.

Se il rango del processo è un multiplo di *(step * 2)* allora esso riceve l'array dal processo con *rango = rango del processo corrente + step.* Se il processo non è un multiplo, allora invia il proprio array al processo con *rango = rango del processo corrente - step.*

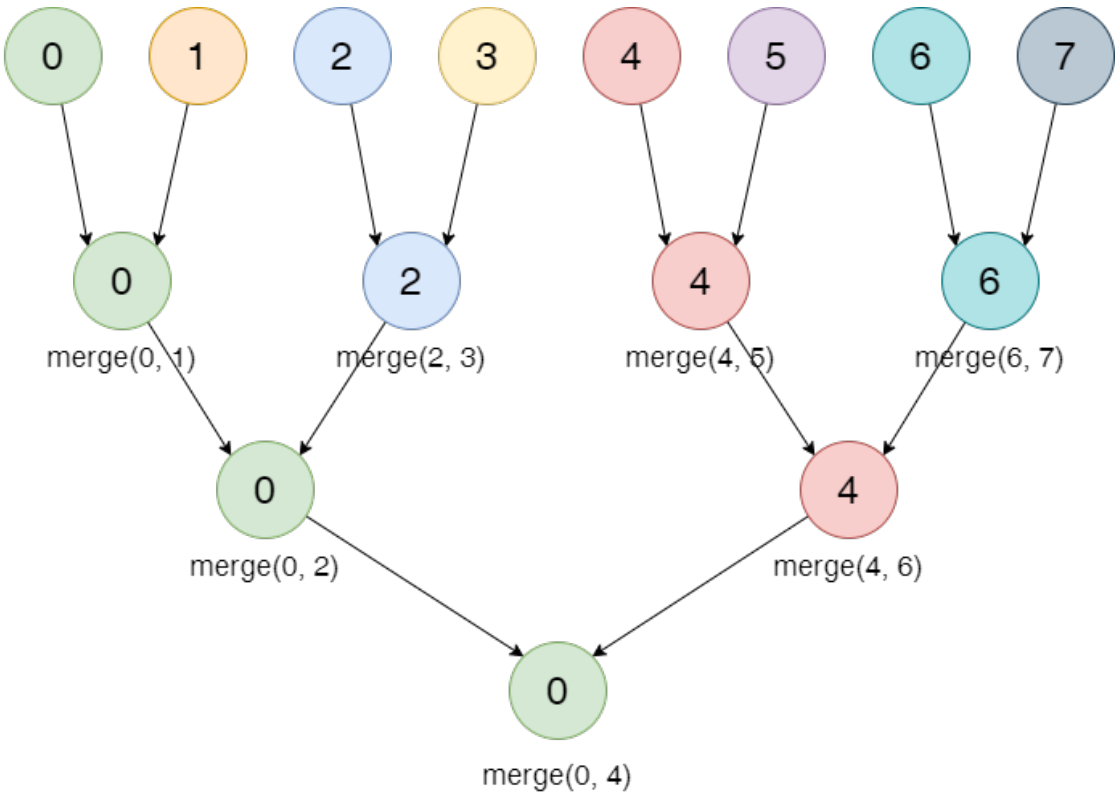
La funzione di merge infine unisce i risultati di due processi in un'unica lista ordinata, che viene sostituita alla lista del processo che ha rango multiplo di *(2 * step).*

Ecco il codice:

```
int step = 1;
while (step < number_of_process){
    if (rank_of_process % (2 * step) == 0){
        if (rank_of_process + step < number_of_process){
            int parter_chunk_size;
            int* parter_chunk;
            MPI_Recv(&parter_chunk_size, 1, MPI_INT, rank_of_process + step, 0, MPI_COMM_WORLD, &status);
            parter_chunk = new int[parter_chunk_size];
            MPI_Recv(parter_chunk, parter_chunk_size, MPI_INT, rank_of_process + step, 0, MPI_COMM_WORLD, &status);
            chunk = merge(chunk, own_chunk_size, parter_chunk, parter_chunk_size);
            own_chunk_size += parter_chunk_size;
            delete[] parter_chunk;
        }
    }
    else{
        MPI_Send(&own_chunk_size, 1, MPI_INT, rank_of_process - step, 0, MPI_COMM_WORLD);
        MPI_Send(chunk, own_chunk_size, MPI_INT, rank_of_process - step, 0, MPI_COMM_WORLD);
        break;
    }
    step = step * 2;
}
```

## 2.2 Dipenda tra i processi

Ad ogni step il processo ricevente deve attendere il risultato parziale del proprio "vicino". In particolare se il rango del processo è un multiplo di *(step * 2)* allora esso deve attendere la terminazione della computazione e l'invio del risultato da parte del processo con *rango = rango del processo corrente + step.*



## 2.3 Tipologia di messaggistica utilizzata

La tipologia di messaggistica utilizzata è **blocking**, in quanto ad ogni passo ogni processo, prima di eseguire l'opportuna operazione di merging, deve aver ricevuto preventivamente gli elementi del suo "vicino". A tale scopo è stata utilizza la funzione ***MPI_Recv*** per i processi con rango multiplo di *(step * 2)* e la funzione ***MPI_Send*** per gli altri.

Non avrebbe avuto senso quindi effettuare un invio/ricezione non bloccante, in quanto nel frattempo il processo non avrebbe comunque potuto svolgere nessuna operazione utile.

## 2.4 Performance

Tempi di esecuzione:

- Seriale

```
c:\Users\Utente\Documents\Università\Anno Accademico
ti\Progetto\Serial>.\serialQSort
TIME: 4.879 secondi
```

- Parallelo:

```
c:\Users\Utente\Documents\Università\Anno Accademico
ti\Progetto\Parallel>mpiexec -n 8 parallelQSort.exe
TIME: 0.816713 secondi
```

Lo **Speed-Up** è pari a: $Ts/Tp = 4{,}88/0{,}82 = \mathbf{5{,}95}$. Lo Speed-Up è quasi pari a 6: essendo il numero di processi utilizzati pari a 8 si tratta di un buon risultato.

L'**efficienza** è pari a: $S/p = 5{,}95/8 = \mathbf{0{,}74}$. Si tratta di un buon valore.

L'**Overhead** è pari a: $To = p*Tp - Ts = 8*0.81 - 4{,}88 = \mathbf{1{,}6}$. Abbiamo quindi un overhead di 1,6 secondi. Il valore è basso in quanto con il merge basato sull'albero, ogni processo invia la sua sotto-sequenza ordinata al suo vicino e ad ogni step viene eseguita un'operazione di merge. Una scelta sbagliata sarebbe stata quella di far raccogliere le sotto-sequenze ordinate al processo master, che avrebbe poi eseguito il merge da solo, chiamando la funzione di merge *p-1* volte. Questa scelta avrebbe degradato le performance.

Si nota infine dalla seguente figura che la versione parallela implementata è molto **scalabile**, in quanto si ottiene un aumento proporzionale della velocità con l'aumento dei processi.

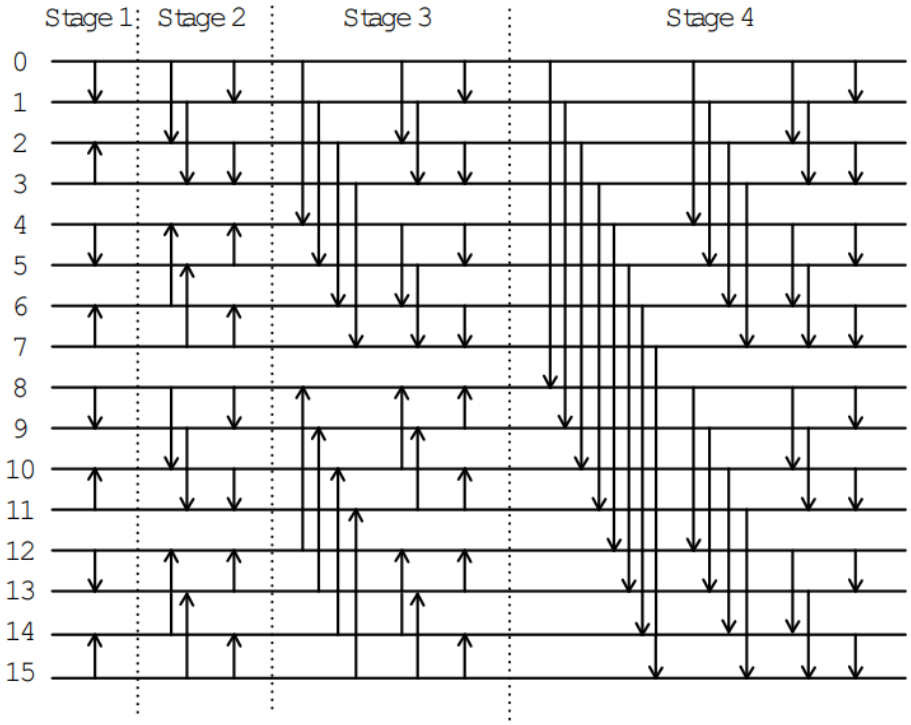
Tuttavia si giunge ad un limite per cui l'overhead causato dalle troppe comunicazioni tra tali processi non giustifica l'aumento degli stessi.

```
c:\Users\Utente\Documents\Università\Anno Accademico 2021
ti\Progetto\Parallel>mpiexec -n 1 parallelQSort.exe
TIME: 4.89732 secondi

c:\Users\Utente\Documents\Università\Anno Accademico 2021
ti\Progetto\Parallel>mpiexec -n 2 parallelQSort.exe
TIME: 1.8682 secondi

c:\Users\Utente\Documents\Università\Anno Accademico 2021
ti\Progetto\Parallel>mpiexec -n 4 parallelQSort.exe
TIME: 0.992907 secondi

c:\Users\Utente\Documents\Università\Anno Accademico 2021
ti\Progetto\Parallel>mpiexec -n 8 parallelQSort.exe
TIME: 0.839564 secondi

c:\Users\Utente\Documents\Università\Anno Accademico 2021
ti\Progetto\Parallel>mpiexec -n 10 parallelQSort.exe
TIME: 0.827832 secondi

c:\Users\Utente\Documents\Università\Anno Accademico 2021
ti\Progetto\Parallel>mpiexec -n 12 parallelQSort.exe
TIME: 0.817321 secondi

c:\Users\Utente\Documents\Università\Anno Accademico 2021
ti\Progetto\Parallel>mpiexec -n 16 parallelQSort.exe
TIME: 0.7733 secondi

c:\Users\Utente\Documents\Università\Anno Accademico 2021
ti\Progetto\Parallel>mpiexec -n 20 parallelQSort.exe
TIME: 0.80123 secondi
```

# 4. Parallel Bitonic Sort

Il Bitonic Sort produce una sequenza ordinata dopo alcuni stage di bitonic merge, che converte due sequenze bitoniche di dimensioni *n* in una sequenza monotona di dimensioni *2n,* attraverso operazioni di **compare-and-exchange.**

Una sequenza bitonica è una sequenza di numeri $a_0, \dots, a_i, a_{i+1}, \dots, a_{n-1}$ con la proprietà che esiste un indice $i$ $(0 \leq i \leq n-1)$, per il quale la sotto-sequenza da $a_0$ a $a_i$ è monotona crescente, mentre la sequenza da $a_{i+1}$ a $a_{n-1}$ è monotona decrescente.

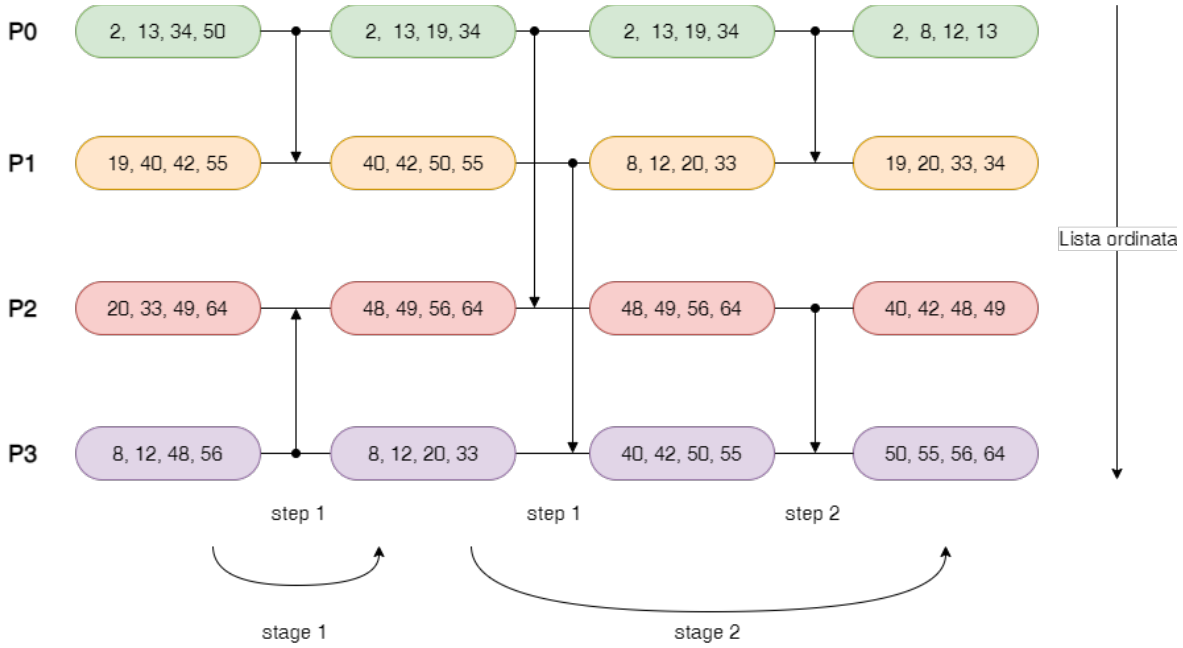



## 3.1 Parallelizzazione e suddivisione del lavoro tra i processi

È possibile mappare il Bitonic Sort su processi paralleli rifacendosi alla sorting network: ogni comparatore è sostituito da una coppia di processi che eseguono un'operazione di **compare-and-exchange.**

Ad ogni processo quindi viene assegnata una parte della lista da ordinare, ovvero *n/p* elementi (dove *n* sono gli elementi totali e *p* i processi) che il processo ordina ***indipendentemente*** utilizzando un algoritmo di ordinamento seriale. Ad ogni **step** ogni coppia di processi effettua un'operazione di compare-exchange e ogni processo mantiene rispettivamente gli *n/p* elementi più piccoli e gli *n/p* elementi più grandi, in accordo con la loro *polarità,* descritta dalla sorting network.

La seguente immagine mostra il processo seguito dal bitonic sort parallelo di 16 numeri con 4 processi.




Inizialmente ogni processo possiede localmente una sequenza ordinata di 4 elementi. Successivamente, attraverso 2 stage, ovvero $log\ (4)$, di operazioni di **compare-and-exchange,** la lista risulta ordinata.

## 2.2 Dipenda tra i processi

Ad ogni stage, e in particolare ad ogni step, la dipendenza tra le coppie di processi varia, a seconda dell'operazioni di compare-exchange da eseguire. Tale dipendenza è quindi funzione dello stage e dello step corrente.

## 2.3 Tipologia di messaggistica utilizzata

La tipologia di messaggistica utilizzata e la **blocking**, in quanto ogni processo prima di eseguire l'opportuna operazione di compare-exchange deve aver ricevuto preventivamente gli elementi del suo partner. A tale scopo è stata utilizza la funzione ***MPI_Sendrecv.***

Non avrebbe avuto senso quindi effettuare un invio/ricezione non bloccante, in quanto nel frattempo il processo non avrebbe comunque potuto svolgere nessuna operazione utile.

## 2.4 Performance

Tempi di esecuzione:

- Seriale

```
C:\Users\Utente\Documents\Università\Anno Accademico 2021
ti\Progetto\Serial>.\serialBitonicSort
TIME: 13.14 secondi
```

- Parallelo:

```
C:\Users\Utente\Documents\Università\Anno Accademico 2021
ti\Progetto\Parallel>mpiexec -n 4 parallelBitonicSort.exe
TIME: 5.43587 secondi
```

Lo **Speed-Up** è pari a: $Ts/Tp$ = 13,14/5,43 = **2,42**. Lo Speed-Up è quasi pari a 2,5: essendo il numero di processi utilizzati pari a 4 si tratta di un risultato discreto.

L'**efficienza** è pari a: $S/p$ = 2,42/4 = **0,60**. Si tratta di un buon valore.

L'**Overhead** è pari a: $To = p*Tp - Ts$ = 4*5,43 – 13,14 = **8,6**. Abbiamo quindi un overhead di quasi 9 secondi. Il valore è abbastanza alto in quanto ogni processo deve comunicare, per ogni step, il proprio chunck di elementi al proprio partner. Avendo eseguito i test su un numero totale di elementi elevato, tale comunicazione richiede del tempo, in quanto il chunck è di grandi dimensioni.

Si nota infine dalla seguente figura che la versione parallela implementata è molto **scalabile**, in quanto si ottiene un aumento proporzionale della velocità con l'aumento dei processi.

Tuttavia si giunge ad un limite per cui l'overhead causato dalle troppe comunicazioni tra tali processi non giustifica l'aumento degli stessi.

```
C:\Users\Utente\Documents\Università\Anno Accademico 2021 -
ti\Progetto\Parallel>mpiexec -n 1 parallelBitonicSort.exe
TIME: 14.8752 secondi

C:\Users\Utente\Documents\Università\Anno Accademico 2021 -
ti\Progetto\Parallel>mpiexec -n 2 parallelBitonicSort.exe
TIME: 8.71999 secondi

C:\Users\Utente\Documents\Università\Anno Accademico 2021 -
ti\Progetto\Parallel>mpiexec -n 4 parallelBitonicSort.exe
TIME: 5.74293 secondi

C:\Users\Utente\Documents\Università\Anno Accademico 2021 -
ti\Progetto\Parallel>mpiexec -n 8 parallelBitonicSort.exe
TIME: 4.29821 secondi

C:\Users\Utente\Documents\Università\Anno Accademico 2021 -
ti\Progetto\Parallel>mpiexec -n 16 parallelBitonicSort.exe
TIME: 3.85027 secondi

C:\Users\Utente\Documents\Università\Anno Accademico 2021 -
ti\Progetto\Parallel>mpiexec -n 32 parallelBitonicSort.exe
TIME: 3.89285 secondi

C:\Users\Utente\Documents\Università\Anno Accademico 2021 -
ti\Progetto\Parallel>mpiexec -n 64 parallelBitonicSort.exe
TIME: 5.63922 secondi
```